# Programming Standards Document

## Checkout and Launch Control Systems (CLCS)

## 84K07500-010

Approval:

| | | | |
|---|---|---|---|
| Chief, Hardware Design Division | Date | Chief, System Engineering and Integration Division | Date |
| Chief, Software Design Division | Date | CLCS Project Controls Office | Date |
| Chief, System Applications Division | Date | Project Manager, CLCS | Date |

**PREPARED BY:** _____

_____

_____

_____

_____

_____

_____

_____

_____

# REVISION HISTORY

| REV | DESCRIPTION | DATE |
|---|---|---|
| | | |

| LIST OF EFFECTIVE PAGES | | | | |
|---|---|---|---|---|
| **Dates of issue of change pages are:** | | | | |
| **Page No.** | **A or D*** | **Issue or Change No.** | **CR No.** | **Effective Date**** |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

# Table of Contents

# PROGRAMMING STANDARD INTRODUCTION

# CHECKOUT AND LAUNCH CONTROL SYSTEMS (CLCS)

## 1. INTRODUCTION

Consistent program style is necessary to improve maintainability, portability and to reduce errors. This standard provides the rules to aid in reaching this consistency across the software development organizations. These rules shall be followed without exception, and are clearly identified in this document. Also included are recommendations which are strong suggestions of style guidelines designed to further support the rules and which are clearly identified in this document. Ultimately, the goal of these standards is to provide the coding style rules and recommendations to increase portability, reduce maintenance and above all improve clarity in a single reference.

### 1.1 PURPOSE

The purpose of this standard is to define one style of programming in C/C++ for CLCS. The standards and recommendations presented here should serve as the basis for continued software development in these languages. This collection of standards should be viewed as a living document; and will be updated as required.

All software shall adhere to the current revision of the standards document at the time of software development. Revisions to the standards document shall not drive CLCS software updates unless explicitly directed.

Software developed to these standards and should be correct and easy to maintain. In order to attain these goals, the programs should:

a) Have a consistent style
b) Be easy to read and understand
c) Be maintainable by different programmers
d) Be portable to other architectures/platforms

### 1.2 SCOPE

This standard applies to all Software developed or maintained for use in the Checkout Launch Control System (CLCS) project at the Kennedy Space Center. The standard covers programming style including naming conventions, program layout, syntax style and class (C++) design consideration. Program design and architecture are beyond the scope of this document.

This document contains both rules and recommendations on software programming practices which are clearly identified as such. Sections designated as rules must be adhered to without exception, unless specified in this document. Recommendations are provided as a guide for

developing all software in as common a style as possible without imposing unnecessary restrictions. Adherence to the recommendations is suggested, but is not mandatory.

## 1.3 AUTHORITY

This document is controlled by the Checkout and Launch Control System (CLCS) Project Manager or the appointed representative.

## 1.4 APPLICABILITY

This standard applies to all Real Time Control (RTC) Application and System Software developed for the CLCS.

## 1.5 STANDARDS MODIFICATION

A form for requesting new rules or changes to the rules of this standard has been included as an appendix to this document. All requests must be submitted to the Software Architecture Team or the appointed representative for evaluation.

## 1.6 REFERENCE RESOURCES

The following references were used in compiling the rules and recommendations of this standard:

Software and Automation Systems Branch C++ Programming Style, Version 1.0
Automation Systems Branch (Code 522), Goddard Space Flight Center, July 1992.

Programming in C++, Rules and Recommendations
Ellemtel Telecommunication Systems Laboratories, Alvsjo, Sweden, 1992.

Recommended C Style and Coding Standards
Bell Labs, Zoology Computer Systems, University of Toronto, CS University of Washington, 1989.

Writing Solid Code
Steve Maguire, Microsoft Press, Redomond, Washington, 1993.

C++ Programming Style
Tony Cargill, Addison-Wesley Publishing, Reading, Massachusetts, 1992.

# 2. NAMING CONVENTIONS

This section of the CLCS Software Standard defines the alphanumeric system for assigning names to the software used to monitor and control the Space Shuttle, its elements and related Ground Support Equipment (GSE) from the CLCS.

## 2.1 NAME CONTROL

The KSC NASA and contractor organizations responsible for the development of software shall also be responsible for the tracking of all program names. A register of all assigned names shall be maintained and no program name shall be reused once it has been issued. The mechanism for assigning program names shall be at the discretion of the responsible Engineering groups.

## 2.2 EXECUTABLE PROGRAM NAMING CONVENTION

***RULE:*** All software executable program names shall be a maximum of 32 characters in length including the extension.

***RULE:*** Two fields shall be used to define the owner and purpose of the program as defined in the following sections. The program name shall be formatted as follows:

**SYS_[Alpha/Numeric Identifier]**

### 2.2.1 System Field

***RULE:*** The first three characters of the name field shall identify the system associated with the program followed by an underscore delimiter. The system fields defined by the responsible engineering groups are as follows:

**RTC Application Software Acronyms**

The acronyms for Application Software CSCI's can be found on the CLCS web site by following the CSCI link from the home page.

**CLCS System Software CSCI Acronyms**

The acronyms for System Software CSCI's can be found on the CLCS web site by following the CSCI link from the home page.

### 2.2.2 Alpha/Numeric Identifier Field

***RULE:*** The alpha/numeric identifier field shall be a unique maximum 28 character string for the program (excluding the extension).

***RECOMMENDATION:*** This field may be used to further define the functionality of the software. It may be a functional description of the software and have unique subsystem identification traits. Each engineering system should determine the character pattern that best fits their system, and document the character pattern template in the requirements. Valid

characters for this field shall include the upper and lowercase alpha characters (***A-Z, a-z***), numerals 0 through 9 (***0-9***), underscore(_), ampersand (***&***) and period (**.**).

## 2.3 FILE NAMING CONVENTIONS

The purpose of these conventions is to provide a uniform interpretation of file names. One reason for this is that it is easier to develop and/or use tools which base their behavior on file name extensions.

### 2.3.1 Required Files

***RULE:*** Each class shall be represented by two source code files: the interface definition (or header) file and the implementation file.

***Approved Exception:*** Two closely interrelated classes may be combined into a single pair of header and implementation files.

### 2.3.2 File Names

***RECOMMENDATION:*** Always give a file a name that is unique in as large a context as possible. Since class names must generally be unique within a large context, it is appropriate to utilize this characteristic when naming its implementation and header files. This convention makes it easy to locate a class definition using a file-based tool.

***RULE:*** A header file shall have the same name as its associated implementation file (e.g. **BaseClass.C** and **BaseClass.h**)

***RULE:*** Interface definition (header) files shall not have filenames that have a system complement (e.g. **"math.h"** and **<math.h>**).

Since the quotes and brackets define the search path for the file, the statement***#include "math.h"*** will include the standard library math include file if the intended one is not found in the current directory.

### 2.3.3 Source Code Filename Extensions

***RULE:*** The extensions listed in the following table shall be used for all source code files.

**CLCS Extensions for Filenames**

| | | | |
|---|---|---|---|
| **<fn>.a** | Archive or library file | **<fn>.mak** | Make file |
| **<fn>.b** | Static binary file | **<fn>.m1** | SL executable file |
| **<fn>.c** | C source code file | **<fn>.man** | Manual page |
| **<fn>.C** | C++ source code file | **<fn>.mpp** | Microsoft Project |
| **<fn>.cpp** | C++ file | **<fn>.o** | Object file |
| **<fn>.csh** | C-shell script | **<fn>.pdf** | Adobe Acrobat file |
| **<fn>.doc** | MS Word document | **<fn>.pl** | Perl file |
| **<fn>.ksh** | K-shell script | **<fn>.ppt** | Powerpoint file |
| **<fn>.g** | SL source file | **<fn>.ps** | Postscript file |

| | | | |
|---|---|---|---|
| **<fn>.gif** | GIF graphic file | **<fn>.s** | Assembly language file |
| **<fn>.h** | Header or include file | **<fn>.sh** | Bourne shell script file |
| **<fn>.htm** | HTML file | **<fn>.so** | Shared Object file |
| **<fn>.html** | HTML file | **<fn>.t** | Manual page |
| **<fn>.java** | JAVA file | **<fn>.txt** | Static text file |
| **<fn>.jpg** | JPEG graphic file | **<fn>.xls** | MS Excel document |

### 2.3.4 LIBRARY FILE NAMING CONVENTIONS

***RECOMMENDATION:*** Library filenames should be unique in as large a context as possible. The name should reflect the function of the library, as well as indicate that it is a library.

## 2.4 CLASS / FUNCTION / VARIABLE NAMING CONVENTIONS

This section defines the conventions for naming classes, functions and variables. When naming a class, function or variable, the names should be as clear as possible. The goal should be to make the user's interface conceptually transparent. The code will be more understandable and readable when names are closely related with their associated purpose.

### 2.4.1 Naming Conventions

***RULE:*** All names shall be mixed case with the initial letter of each word capitalized (e.g. BaseClass), with further distinction by Type as defined with the exception of constants.

***RULE:*** Names that differ only by the use of upper-case and lower-case letters shall not be used.

***RULE:*** Underscores shall not be used in any user application name.

***RULE:*** Preface pointer variables with the character '***p***'.
The difference between a pointer, object and a reference to an object is important for understanding the code, e.g. **int** *pNumber;.

***RULE:*** Preference reference variables with the character '***r***'.
The difference between variable types is clarified. This establishes the difference between a method returning a modifiable object and the same method name returning a non-modifiable object, e.g. Status& rStatus;.

***RULE:*** The following six conventions shall be followed for all class, function and variable names.

1. Abstract base classes shall begin with a capital 'V' with the first letter of each word capitalized thereafter (e.g. **VBaseClass**).
2. Function names shall be mixed case with the initial letter of each word capitalized (e.g. **OpenValve**).
3. Class member attributes shall begin with a lower-case 'm' with the first letter of each word capitalized thereafter (e.g. **mMemberObject**).

4. Global variables (which should be avoided if at all possible) shall begin with a lower-case ‘g’ with the first letter of each word capitalized thereafter (e.g. **gGlobalVariable**).
5. Local variables shall begin with the first word in lower-case with the first letter of each word capitalized thereafter (e.g. **internalVariable**).
6. Constants (***const*** and ***enum***) and macros shall be named in all upper-case letters (e.g. **MAXPRESS**).

### 2.4.2 Naming Recommendations

The following conventions should be followed when naming classes, functions and variables. Adherence to these suggestions will greatly enhance the maintainability of the software.

***RECOMMENDATION:*** Choose names that suggest the usage. One rule of thumb is that a name which cannot be pronounced is a bad name. A long name is normally better than a short, cryptic one, but the truncation problem must be taken into consideration. Abbreviations can always be misunderstood. Global variables, functions and constants should have long enough names to avoid name conflict, but not unreasonably long.

***RECOMMENDATION:*** Names should not include abbreviations that are not generally accepted.

# 3. PROGRAMMING RULES AND RECOMMENDATIONS

This section identifies the format to be used for CLCS Application Software that is not automatically generated.

## 3.1 FILE ORGANIZATION

### 3.1.1 Source Code Contents

***RULE***: Each CLCS source code file shall contain nine sections of data in a specified order. For consistency, each section shall be identified by a comment containing the section. If a section does not apply, a comment shall be included following the title which indicates "This section not applicable".

***RULE***: Each source code file shall have the following ordered sections preceded by a section title. The sections are defined as:

1. Introductory Comment containing:
   /* Introduction */
   - Name: Function name
   - Description: A brief description of the purpose
   - Notes: Refer to Appendix A-1 for a description
   - Warnings/Limitations: Refer to Appendix A-1
   - Revision History: Most recent revision is at the top and includes the Ending Revision, WAD and description of the change, Programmer, Organization, and date.
   - Copyright (Year)National Aeronautics and Space Administration, All rights reserved. Where Year represents the year the baseline revision was created.

2. System include files (e.g. <iostream.h>)
3. Application specific include files (e.g. "VBaseApplication.h")
4. External functions
5. External variables
6. Constants
7. Macros
8. Class Declarations
9. Non-member functions

### 3.1.2 File Size

***RECOMMENDATION:*** File size should be limited to approximately 1000 lines of source code. Although there is no real maximum length for source files, those with more than 1000 lines are cumbersome to deal with. Editors may not have enough temporary space to edit the file and compilations will go slower.

### 3.1.3 Line Length

***RECOMMENDATION:*** Line lengths should be limited to 79 columns. Lines longer than 79 columns are not handled well by all terminals and should be avoided if possible. Excessively long lines which result from deep indentation are often a symptom of poorly organized code.

Source code lines should not wrap unless absolutely necessary, as in the definition of a long literal. This can lead to confusion about program structure.

### 3.1.4 Include File Contents

***RULE:*** An application include file shall contain only the necessary include files to support the classes referenced in the file. It is more maintainable to have all supporting files automatically included when a class include file is used than requiring the user to remember to include those support files.

***RULE:*** A **.h** file shall include only the files required to compile the **.h** file.
Extraneous .h files will increase compilation time. This will be significant considering .h files are frequently compiled. Unlike .C files which are compiled to object code and then linked, .h files are compiled every time they are included. Some compiler venders combat this problem with pre-compiled headers where the .h file, when successfully compiled, produces an output file that contains symbol table information. Subsequent compilations that include the .h file will save time by utilizing the previously generated symbol table information rather than regenerating it.

### 3.1.5 Multiple Inclusion Prevention

***RULE:*** Every include file shall contain a mechanism that prevents multiple inclusions of the file.

***RECOMMENDATION:*** The easiest way to avoid multiple file includes is to use a ***#ifndef / #define*** block at the beginning of the file and ***#endif*** at the end of the file.

## 3.2 SOURCE CODE COMMENTS

***RECOMMENDATION:*** It is essential to document source code to capture the thought processes behind the software development. It may also be helpful to include a requirements tracking cross-reference. This should be compact and easy to find. By properly choosing names for variables, functions and classes and by properly structuring the code, there is less need for lengthy comments within the code. An example of the function comment header is provided in Appendix A-2.

Note that comments in include files are meant for the users of classes, while comments in implementation files are meant for those who maintain the code. A tactical comment describes what a single line of code is intended to do and is placed, if possible, at the end of the line. Unfortunately, too many tactical comments can render the code unreadable.

### 3.2.1 Comment Maintenance

***RULE:*** Comments shall be maintained with the same attention as the associated code.

***RECOMMENDATION:*** It is extremely important to keep comments in synchronization with the code. Comments that disagree with the code are of negative value.

### 3.2.2 Comment Syntax

***RECOMMENDATION:*** Use the "//" syntax for all comments (C++ compilers only). If the characters "//" are used consistently for writing comments, then the combination of "**/* */**" may be used to make comments out of entire sections of code during development and debug phases. An example of the section comment header is provided in Appendix A-3.

### 3.2.3 Strategic Comments

A strategic comment describes what a function or section of code is intended to do and is normally placed before the code.

***RULE:*** All strategic comments shall appear before the section of code they document.

***RULE:*** Every function shall contain a strategic comment before the function declaration explaining its purpose and anything special about the function.

***RECOMMENDATION:*** Putting a comment at the top of a 3 - 10 line section explaining the purpose of the code is often more useful than a comment on each line describing the micrologic. It is also helpful to add a comment for the parameter list which describes the purpose of each parameter.

## 3.3 OPTIMIZATION

Program performance may be enhanced or improved through the process of optimization. The application should be studied to identify the important measures, i.e. worst case timing and operation frequencies. Assess the trade-offs of alternative implementations as they relate to memory, speed, and simplicity of implementation prior to making optimization modifications. This section provides the recommendations to be followed for code optimization.

### 3.3.1 Condition Assertion

***RECOMMENDATION:*** The ***assert*** macro (contained in **<*assert.h*>**) should be used whenever possible to assist in the detection and isolation of errors. ***assert*** is a debug-only macro that aborts execution if its argument is false. ***assert*** should not disturb memory or initialize data that would otherwise be uninitialized or cause any other side effects.

### 3.3.2 Performance Enhancements

***RECOMMENDATIONS:*** Optimize code only if it has a known performance problem. Industry tests have demonstrated that a lot of time is spent optimizing code that is seldom or never executed. It is important to identify and modify only the portions of the code that are affected since optimization compromises extensibility, reusability, and understandability.

Performance measurement development tools (e.g. Pure Software, Quantify, CenterLine TestCenter) should be used to provide performance measurements that can be used to improve performance and isolate run-time errors.

## 3.4 PORTABILITY

C/C++ code is not inherently portable. Thought and effort are required to make it so. As a general practice, all other things being equal, portable code is better than non-portable code. This section provides the rules and recommendations for portability standardization.

### 3.4.1 Main Routine Form

***RULE:*** The heading for main shall be defined as either:

***int main (void)***
***int main (int argc, char *argv[ ])***

These are the forms explicitly sanctioned by the ANSI-C standard. Other common forms, such as
***void main (void)*** only work on some platforms.

### 3.4.2 Program Exit Codes

***RULE:*** The constants ***EXIT_SUCCESS*** and ***EXIT_FAILURE*** shall be used as program exit codes.

Calling ***exit(0)*** is portable, but ***exit(1)*** is not. A return statement in main effectively calls exit, so main should normally return ***EXIT_SUCCESS***. These constants are defined in ***<stdlib.h>.***

### 3.4.3 Object Size Determination

***RULE:*** The generic type ***size_t*** (defined in several standard headers) shall be used as the type of an object that holds the size of another object.

***size_t*** is always defined as an unsigned integer type, which varies from platform to platform. ***unsigned long int*** could be used in place of ***size_t***, but it may be wasteful on some architectures. ***size_t*** is always the right size (e.g. ***size_t strlen (const char*)***).

### 3.4.4 Pointer Difference Calculation

***RULE:*** When the difference between two pointers is required, the value shall be stored in an object of type ***ptrdiff_t*** (defined in <***stddef.h***>).

Each implementation defines ***ptrdiff_t*** as a signed integer type of the appropriate size for the target architecture.

### 3.4.5 User Unique Data

***RECOMMENDATION:*** The generic types of ***void*** and ***void(*)()*** should be used for user unique data objects and functions. The ***void*** type is guaranteed to have enough bits of precision to hold a pointer to any data object. The ***void(*)()*** type is guaranteed to be able to hold a pointer to any function. Be sure to cast pointers back to the correct type before using them.

### 3.4.6 Int Data Type

***RECOMMENDATION:*** Be very careful defining data structures using the ***int*** data type. If a stream moves data between platforms with different base register sizes (e.g. HP uses 32-bit, DEC Alpha uses 64-bit), the effective size of the ***int*** will cause data structures to be misaligned.

### 3.4.7 System Calls

***RULE:*** System calls shall not be used that are platform dependent.

### 3.4.8 External File References

***RULE:*** Always specify the full pathname for all files and/or programs used.

***RECOMMENDATION:*** When referencing external files and/or programs, specify the path using either a configuration file or environment variable. Avoid hard-coded pathnames. The use of hard-coded pathnames will cause the re-code of modules when the file structure changes. Also, do not assume a user's ***PATH*** variable includes the directory for what should be a common file.

### 3.4.9 Temporary Files

***RECOMMENDATIONS:*** When using temporary files, use the UNIX function ***tempnam*** to ensure unique filenames.

Use a directory other than ***/tmp*** for temporary file storage. On some systems, the /tmp directory usage is restricted by the System Administrator. Using this directory to store temporary files in this circumstance will cause the read/write to fail.

# 4. C/C++ CODING STANDARDS

This section of the CLCS Application Software Standard defines the coding standards to be used in the application software.

## 4.1 GENERAL CODING

This section defines the general coding standards which will assist in keeping consistency across the different application sets.

### 4.1.1 Braces Placement

***RULE:*** Braces "{ }" which enclose a code block shall be placed in the same column on separate lines directly before and after the block, in the following:

```
if (condition == TRUE)
{
        fTrueFunction1( );
        fTrueFunction2( );
}
else
{
        fFalseFunction1( );
        fFalseFunction2( );
}
```

### 4.1.2 Parenthesis Placement

***RULE:*** Parentheses shall be placed adjacent to methods, not keywords.
Parenthesis placement will lend to the readability of code, and making it easier to distinguish between a function and a keyword.
E.g. ***Keyword***: *if (expression)*; ***method***: *strcpy(s1,s2);*

### 4.1.3 Operator Usage

***RULE:*** Spaces around the "**.**" and "**->**" operators shall not be used.

***RULE:*** Floating point comparisons shall use the**>=** or **<=** operators.
Floating point comparisons should always be made with the >= and <= operators rather than the equality operators (==, !=) because the underlying representation and precision for floating point values may differ from machine to machine and may not be exact. E.g. 25.38 may be represented as 25.376 or 25.3834.

***RULE:*** The constant shall be placed on the left hand side of a comparison.
E.g. if (constant == variable). If one of the operators is omitted, the error will be caught at compile time. For readability, clarity, and maintenance, the value being checked is the first item in the expression versus being lost at the end of the source code line or the next line.

***RULE:*** Blocks of attribute and variable declarations and initializations shall be vertically aligned to promote clarity and readability.

***RULE:*** Pointer arithmetic must be well documented.
In general, pointers are a cause of numerous software development errors. Any manipulation of pointers should be performed with extreme care and scrutiny. Documenting the purpose for pointer arithmetic provides ease in maintaining the source code.

### 4.1.4 Flow Control Statements

This section defines the rules and recommendations for controlling the flow of program execution.

***RULE:*** Only one statement shall be included per line of source code.

***RULE:*** The code following a case label shall always be terminated by a *break/return* statement.
When several case labels are followed by the same block of code, only one *break/return* statement is required. If the code which follows a *case* label is not terminated by a *break/return*, the execution continues after the next *case* label. This means that poorly tested code can be erroneous and still seem to work.

***RULE:*** A *switch* statement shall always contain a *default* branch which handles unexpected cases.

***RECOMMENDATION:*** The flow control statements *while, for* and *do* should be followed by a code block, even if it is empty.

Although everything that is to be performed in a loop can sometimes easily be written on one line in the loop statement, concluded with a semicolon at the end of the line, this may lead to misreading of the code since the semicolon may be missed. It is better to place an empty block of code following the statement to completely clarify the code.

***RECOMMENDATION:*** Error handling needs to be addressed here.

***RECOMMENDATION:*** Use the *goto* syntax with caution and deliberation.
***goto*** breaks the control flow and can lead to code that is difficult to comprehend and maintain. For extremely time critical applications, *goto* may be permitted. Every such usage must be carefully motivated, and should be explained in a comment.

***RECOMMENDATION:*** Use parenthesis to clarify the order of evaluation for operations in expressions.
If the operator precedence is not obvious or the expression is complicated or long, use parenthesis to make it more readable. Even if the parenthesis are not technically required, make the order of evaluation obvious.

***RECOMMENDATION:*** Check the fault codes which may be received from library functions even if the function seems foolproof.

Two important characteristics of a robust system are that all faults are reported and, if the fault is so serious that continued execution is not possible, the process is terminated. In this way the propagation of faults through the system is avoided. In achieving this, it is important to always test fault codes from library functions (e.g. opening/closing files, allocation of memory for data). One test too many is better than one test too few. Application specific functions should preferably not return fault codes but should instead take advantage of exception handling features.

### 4.1.5 Memory Allocation

***RULE:*** *malloc, realloc* or *free* shall not be used for memory allocation (C++ only).
In C, ***malloc, realloc*** and ***free*** are used to allocate memory dynamically on the heap. This may lead to conflicts with the use of the *new* and *delete* operators in C++. It is dangerous to:

1. Invoke ***delete*** for a pointer obtained via ***malloc/realloc***.
2. Invoke ***malloc/realloc*** for objects having constructors.
3. Invoke ***free*** for anything allocated using ***new***.

***RULE:*** Empty brackets "[ ]" shall always be provided for *delete* when deallocating arrays (C++ only).
If an array 'p' having a type 'T' is allocated, it is important to invoke ***delete*** in the correct way:

1. (WRONG) ***delete p*** results in the destructor being invoked only for the first object of type T.
2. (WRONG) ***delete [m] p*** where 'm' is an integer which might be greater than the number of objects allocated earlier, the destructor for 'T' will be invoked for memory that does not represent objects of type T.
3. (CORRECT) ***delete [] p*** is the correct way since the destructor will then be invoked only for those objects which were allocated earlier.

***RULE:*** Memory that is allocated shall be deallocated when it is no longer needed. Do not allocate memory and expect that someone else will deallocate it later. For instance, a function can allocate memory for an object which is then returned to the user as the return value for the function. There is no guarantee that the user will remember to deallocate the memory, and the interface with the function then becomes considerably more complex.

***RULE:*** A pointer that points to deallocated memory shall always be assigned to a new value. Pointers that point to deallocated memory should either be set to 0 or be given a new value to prevent access to released memory. This can be a very difficult problem to solve when there are several pointers which point to the same memory since C++ has no garbage collection.

***RECOMMENDATION:*** Avoid frequently allocating and deallocating memory (C only) which may cause memory fragmentation.

### 4.1.6 Standard Error Handling

Standard Error Handling needs to be addressed here.

## 4.2 CLASS CODING (C++ ONLY)

While reading the following sections, it is important to understand the distinction between a class declaration and a class definition.

- Class Declaration: A class declaration is contained in a .H file and gives the specification of the class.
- Class Definitions: A class definition is contained in a .C file and identifies the class implementation.

***RULE:*** The public, protected and private sections of a class definition shall be declared in that order.
Both member attributes and member functions shall be declared in the same appropriate section. By placing the public section first, everything that is of interest to a user is gathered at the beginning of the class definition. The protected section may be of interest to designers when considering inheriting from the class. The private section contains details that should have the least general interest.

***RECOMMENDATION:*** Make classes as simple as possible.
Give each class a clear purpose. If classes grow too complicated, make more classes: break complex classes into simpler ones.

***RECOMMENDATION:*** *Friend*s of a class should be used to provide additional functions that are best kept outside of the class.
A *friend* is a non-member of a class that has access to the non-public members of the class. Friends offer an orderly way of getting around data encapsulation for a class. Friends are good if used properly, but the use of many friends can indicate that the modularity of the system is poor.

***RULE:*** Multiple inheritance shall not be used.
It is for getting out of bad situations, especially repairing interfaces where control over the broken class belongs to someone else. The complexities of multiple inheritance override its usefulness in most situations.

***RECOMMENDATION:*** Polymorphism needs to be addressed here. No downcasting should be used.

### 4.2.1 Required Class Functions

***RULE:*** All class definitions shall have the constructor, destructor and assignment operator (*operator*) defined.
Don't let the compiler create these functions. Class designers should always say exactly what the class should do and keep the class entirely under their control. If a copy constructor or assignment operator is not desired, declare it private. Remember, if any constructor is specified, it prevents the default constructor from being synthesized.

***EXCEPTION:*** The simplest of accessor functions may be defined inline in the class declaration. E.g.:

```
class Example
{
public:
        int getI() const {return (mI);}
        void setI(int I) {mI = I);}
private:
        int mI;
};
```

This reduces the number of lines of code. Across the project the code reduction will be significant, and thereby reduce the cost. This savings should be realized without significantly degrading the readability of the class declarations.

***RULE:*** A class which uses *new* to allocate instances managed by the class shall define a *copy* constructor.
A ***copy*** constructor is recommended to avoid surprises when an object is initialized using an object of the same type. If an object manages the allocation and deallocation of objects on the heap, only the value of the pointer will be copied. This can lead to two invocations of the destructor for the same object, probably resulting in a run-time error.

***RULE:*** All classes which are used as base classes shall define a *virtual* destructor.
If a class having virtual functions but without virtual destructors is used as a base class, there may be a surprise if pointers to the class are used. If such a pointer is assigned to an instance of a derived class and if *delete* is then used on the pointer, only the base class' destructor will be invoked. If the program depends on the derived class' destructor being invoked, it will fail.

***RULE:*** If a base class does not have a virtual destructor, no derived class nor members of a derived class should have a destructor. If a derived class or member of a derived class defines a destructor and the base class destructor remains non-virtual, memory leaks or other abnormalities can occur.

### 4.2.2 Member Function Rules and Recommendations

***RULE:*** Member functions shall only be prototyped within a class definition.
A member function that is defined within a class definition automatically becomes an in-line function. Class definitions are less compact and more difficult to read when they include definitions of member functions. It is easier for an in-line member function to become an ordinary member function if its definition is placed outside of the class definition.

***RULE:*** A member function that does not affect the state of an object shall be declared ***const***.
Member functions declared as ***const*** may not modify member data and are the only functions which may be invoked on a ***const*** object. A ***const*** declaration is excellent insurance that objects will not be modified when they should not be. ***const*** member functions may never be invoked as an "lvalue" (a location value where a value may be stored).

***RULE:*** A public member function shall never return a non-***const*** reference or pointer to member data.
By allowing a user direct access to the private member data of an object, this data may be changed in ways not intended by the class designer.

***RULE:*** Inside a constructor, *only* initialize variables and/or actions which cannot fail.
Create initialization routines to complete construction. Constructors cannot return an error, therefore object instantiators must check an object for errors after construction, which is often forgotten. Throwing exceptions from a constructor may leave the object in an inconsistent state.

***RULE:*** All functions shall have a prototype definition.
This practice helps eliminate function calling errors that might otherwise have been avoidable. This purpose can be strengthened by making the argument types more accurate (e.g.***unsigned ch*** instead of ***int***). The drawback to this is that it may be necessary to cast arguments to silence noncritical type mismatch warnings.

***RULE:*** The names of formal arguments to functions shall be specified and are to be the same in both the function declaration and in the function definition.
The names of formal arguments may be specified in both the function declaration and definition in C++, even if these are ignored by the compiler in the declaration. Providing for function arguments is part of the function documentation. The name of the arguments may clarify how they are used, reducing the need to include comments for documenting that purpose.

***RULE:*** The return type of a function shall always be provided explicitly.
If no return type is explicitly provided, it is, by default, an*int*. To improve the readability and clarity of the code, function return types must be specified.

***RULE:*** A public function shall never return a reference or pointer to a local variable.
If a function returns a reference or pointer to a local variable, the memory to which it refers will already have been deallocated when the reference or pointer is used. The compiler may or may not give a warning of this.

***RULE:*** When two operators are opposite (such as "==" and "!="), both shall be defined even if only one is necessary.
Define one in terms of the other, e.g.

```
bool
Example::operator == (Example&  rhs)
{
  // Test to see if the rhs (right hand side) is
  //  equal to "this" instance.
  …
  return (retValue);
}
```

```
bool
Example::operator != (const Example&  rhs)
{
  return (!(this == rhs));
}
```

***RECOMMENDATION:*** When declaring functions, the leading parenthesis and the first argument (if any) should be written on the same line as the function name.
If space permits, other arguments and the closing parenthesis may also be on the same line. Otherwise, each additional argument should be written on a separate line (with the closing parenthesis directly after the last argument).

***RECOMMENDATION:*** Avoid functions with many arguments.
Functions having long lists of arguments look complicated, are difficult to read, and can indicate poor design. In addition, they are difficult to read and to maintain.

***RECOMMENDATION:*** Pass arguments by *const* reference as the first choice.
As long as the object being passed in does not need to be modified, this practice is best because it has the simplicity of pass-by-value syntax but does not require expensive constructions and destructions to create a local object.

***RECOMMENDATION:*** If a function stores a pointer to an object which is accessed via an argument, the argument should have the type pointer.
Use reference arguments in other cases. By using references instead of pointers as function arguments, code can be made more readable, especially within the function. A disadvantage is that it is not easy to see which functions change the values of their arguments.

***RECOMMENDATION:*** Watch for overloading.
A function should not conditionally execute code based on the value of an argument (default or not). In this case, two or more overloaded functions should be created.

***RECOMMENDATION:*** Use overloaded function names instead of different function names to distinguish between functions that perform the same operations on different data types. Remember, C++ does not permit overloading on the basis of the return type.

***RECOMMENDATION:*** Consider default arguments as an alternative to function overloading. In general, replacing function overloading by a default argument makes a program easier to maintain because there is only one copy of the function body (e.g. function A
(int a, char b, x=1)).

***RECOMMENDATION:*** When overloading functions, all variations should be used for the same purpose. Overloading of functions can be a powerful tool for creating a family of related functions that only differ as to the type of data provided as arguments. If not used properly (such as using functions with the same name for a different purpose), they can, however, cause considerable confusion.

***RECOMMENDATION:*** Avoid long and complex functions.
If a function is too long it can be difficult to comprehend. Generally, it can be said that a function should be no longer than two pages since that is about how much that can be comprehended at one time.

If an error situation is discovered at the end of an extremely long function, it may be difficult for the function to clean-up after itself and to "undo" as much as possible before reporting the error to the calling function. By using short functions, such an error can be more exactly localized. Complex functions are also much more difficult to test.

### 4.2.3 In-Line Member Function Rules and Recommendations

***RULE:*** In-line functions shall not be defined in the interface definition (header) file for the class.

***RECOMMENDATION:*** Access and forwarding functions should be in-line member functions. This will improve the performance of the class.

***RULE:*** Constructors and destructors shall not be defined as in-line functions.
A constructor always invokes the constructors of its base classes and member data before executing its own code. This cascading of in-line constructors may be too complex for some compilers to handle efficiently.

***RULE:*** In-line functions and parameterized types shall be used instead of preprocessor macros.
This allows more parameter checking to be performed at compilation.

***RECOMMENDATION:*** Use of in-line functions in small programs can help performance. Extensive use of in-line functions in large projects can actually hurt performance by enlarging code, causing paging problems and forcing many recompilations.

### 4.2.4 Member Attribute, Variables and Constants

***RULE:*** In variable declarations, the pointer qualifier (*) shall be with the variable name rather than with the type.
Declare variables as ***char *s, *t, *u;*** rather than ***char* s, t, u;*** In the latter instance, only *s* is declared as a character pointer.

***RULE:*** Public member data shall not be specified in a class definition.
A public variable represents a violation of one of the basic principles of object oriented programming, namely, data encapsulation. Access functions should be used to return values of private member data. This avoids the possibility of an arbitrary function changing the public data value which may lead to errors that are difficult to locate.

***RULE:*** Symbolic values shall be defined instead of numeric values in code.
Numerical values in code can be the cause of difficult problems if and when it becomes necessary to change a value. A large amount of code can be dependent on such a value never

changing and the value can be used at a number of places in the code, leading to difficulty in locating all instances of them.

***RULE:*** Constants shall be defined using *const* or *enum* instead of *#define.*
The preprocessor performs a textual substitution for macros in the source code which is then compiled. This can lead to a number of negative consequences. Names declared with ***#define*** are untyped and unrestricted in scope. In contrast, names declared with ***const*** are typed and follow C++ scope rules.

***RECOMMENDATION:*** Variables should be declared with the smallest possible scope.
A variable ought to be declared with the smallest scope possible to improve the readability of the code and so variables are not unnecessarily allocated.

***RULE:*** Every variable that is declared shall be given a value before it is used.
A variable must be initialized before it is used. Normally the compiler gives a warning if a variable is undefined. Instances of a class are usually initialized even if no arguments are provided in the declaration (the empty constructor is invoked). By initializing all variables before they are used, the code is made more efficient since no temporary objects are created for the initialization. For objects having large amounts of data, this can result in significantly faster code. To declare a variable that has been initialized in another file, the keyword ***extern*** is always used.

***RULE:*** Pointers shall not be assigned a value of *NULL* and shall not be compared to *NULL*.
A value of *0* (zero) shall be used instead. According to the ANSI-C standard (and the pending ANSI-C++ standard), ***NULL*** is defined as ***(void *)0*** or as ***0***. This may lead to errors unless an explicit type conversion is supplied.

***RECOMMENDATION:*** Use *unsigned* for variables which cannot reasonably have negative values.

## 4.3 AUTO GENERATED DOCUMENTATION

The DOC++ tool is being used to extract information from the source code to automatically generate software documentation. Refer to the DOC++ Users Guide and examples found in Appendix A.

Need to address code generated by COTS tools vs. Standards here.

# 5. MAKEFILES

The following methods include support for using appropriate compiler flags which ensure portability. It is strongly recommended that the application software standard Makefile template is used.

## 5.1 TARGETS

### 5.1.1 Clean Target

***RULE:*** All Makefiles shall include a ***clean*** target. The ***clean*** target restores the directory to the state it would be in if the source had just been checked-out from the source code library. The Makefile invokes the ***clean*** target for Makefiles in directories below it.

### 5.1.2 All Target

***RULE:*** All Makefiles shall include an ***all*** target.
The ***all*** target makes everything in that directory. The Makefile invokes the ***all*** target for Makefiles in directories below it.

### 5.1.3 Default Target

***RULE:*** All Makefiles shall include a ***default*** target.
The ***default*** target must be the uppermost target in the Makefile. Its sole function is to remind the user to invoke the Makefile through one of the appropriate targets.

### 5.1.4 Makedepend Target

***RECOMMENDATION:*** All Makefiles should include a ***makedepend*** target to assist in the setting of dependencies.
The ***makedepend*** target invokes the ***makedepend*** utility. The Makefile dependencies are then automatically generated and added to the Makefile.

## 5.2 TAGS

***RECOMMENDATION:*** ***Tags*** should be included in the Makefile when using the EMACS editor.
The ***Tags*** function results in EMACS setting flags to identify function names, variable names, etc. When you want to locate the definition of the function or variable, you simply click on the name and EMACS searches for and displays the location of the definition.

## 5.3 MACROS

The following macros shall be defined in all Makefiles:

***RULE:*** The **CC Macro** shall be set to the requested compiler (cc, acc or gcc) along with a switch indicating the optimize or debug level for the compiler.
A switch for ANSI-C and C++ is also available.

***RULE:*** The **LLIBS Macro** gives the link library for C Code.
All libraries required for proper linking shall be included in this macro definition.